[[](http://www.aosabook.org/en/index.html)<>](http://www.aosabook.org/en/index.html)

**Berkeley DB**

[Margo Seltzer](http://www.aosabook.org/en/intro1.html#seltzer-margo) and [Keith Bostic](http://www.aosabook.org/en/intro1.html#bostic-keith) 玛戈・舒尔茨 和凯斯・波斯蒂克

Conway's Law states that a design reflects the structure of the organization that produced it. Stretching that a bit, we might anticipate that a software artifact designed and initially produced by two people might somehow reflect, not merely the structure of the organization, but the internal biases and philosophies each brings to the table. One of us (Seltzer) has spent her career between the worlds of filesystems and database management systems. If questioned, she'll argue the two are fundamentally the same thing, and furthermore, operating systems and database management systems are essentially both resource managers and providers of convenient abstractions. The differences are "merely" implementation details. The other (Bostic) believes in the tool-based approach to software engineering and in the construction of components based on simpler building blocks, because such systems are invariably superior to monolithic architectures in the important "-bilities": understandability, extensibility, maintainability, testability, and flexibility.

康威法则（Conway’s law）说明了设计反映了产生它的组织的结构。展开来说，我们也许会预见一款由两个人设计和完成最初制作的软件不仅会在一定程度上反映组织的结构，还会反映每一位带来的内在偏见和哲学理念。我们中的一位（Seltzer）在文件系统和数据库管理系统的世界中度过她的职业生涯。如果被问及于此，她会辩解说此二者基本上是等同物，进一步地，操作系统和数据库管理系统实质上都既是资源管理器又是便利抽象层的提供者。它们的区别“仅仅”在于实现的细节。另一位（Bostic）则信仰软件工程中基于工具的方法和基于简单构造块的组件构建方法，因为这样的系统在各种重要“能力”方面总是优于单体式体系结构：可理解性、可扩展性、可维护性、可测试性和灵活性。

When you combine those two perspectives, it's not surprising to learn that together we spent much of the last two decades working on Berkeley DB—a software library that provides fast, flexible, reliable and scalable data management. Berkeley DB provides much of the same functionality that people expect from more conventional systems, such as relational databases, but packages it differently. For example, Berkeley DB provides fast data access, both keyed and sequential, as well as transaction support and recovery from failure. However, it provides those features in a library that links directly with the application that needs those services, rather than being made available by a standalone server application.

当把这两种理念结合起来，你就不会奇怪我们花了过去二十年间的大部分时光共事于Berkeley DB（一个提供高速、灵活、可靠和可扩展的数据管理的软件库）了。Berkeley DB提供了人们所期待的传统系统（例如关系型数据库）中的大多数的同样功能，但是打包方式不同。例如，Berkeley DB提供了按键值的和按顺序的两种快速数据访问，同时还有事务支持和故障恢复。但是，它以库的形式提供这些特性，与需要这些服务的应用程序链接到一起，而不是作为一个独立的服务器应用提供服务。

In this chapter, we'll take a deeper look at Berkeley DB and see that it is composed of a collection of modules, each of which embodies the Unix "do one thing well" philosophy. Applications that embed Berkeley DB can use those components directly or they can simply use them implicitly via the more familiar operations to get, put, and delete data items. We'll focus on architecture—how we got started, what we were designing, and where we've ended up and why. Designs can (and certainly will!) be forced to adapt and change—what's vital is maintaining principles and a consistent vision over time. We will also briefly consider the code evolution of long-term software projects. Berkeley DB has over two decades of on-going development, and that inevitably takes its toll on good design.

在本章中，我们将要更深入地观察Berkeley DB，看到它由一组模块组成，每个模块都体现了Unix的“把一件事做好”的哲学。嵌入了Berkeley DB的应用程序能够直接使用这些模块或者通过更加熟悉的操作获取、存放和删除数据项来间接使用它们。我们将集中关注体系结构——我们是如何开始的，我们设计了什么，我们在哪结束了以及为什么。设计能够（而且一定将要）被强迫去适应和改变——重要的是随时间的推移而维护原则和一致的愿景。我们也将简要的谈及长期软件项目的代码演进。Berkeley DB有超过20年的持续开发，这难免会给好的设计造成负面影响。

***4.1. In the Beginning 开端***

Berkeley DB dates back to an era when the Unix operating system was proprietary to AT&T and there were hundreds of utilities and libraries whose lineage had strict licensing constraints. Margo Seltzer was a graduate student at the University of California, Berkeley, and Keith Bostic was a member of Berkeley's Computer Systems Research Group. At the time, Keith was working on removing AT&T's proprietary software from the Berkeley Software Distribution.

Berkeley DB起源于Unix操作系统还专属于AT&T的时代。那时有几百种实用工具和函数库的血统还带有严格的许可限制。Margo Seltzer那时是加州大学伯克利分校的研究生，Keith Bostic是伯克利计算机系统研究组的一员。当时Keith正在从伯克利软件发行版（BSD）中删除AT&T的专属软件。

The Berkeley DB project began with the modest goal of replacing the in-memory hsearch hash package and the on-disk dbm/ndbm hash packages with a new and improved hash implementation able to operate both in-memory and on disk, as well as be freely redistributed without a proprietary license. The hash library that Margo Seltzer wrote [[SY91](http://www.aosabook.org/en/bib1.html#bib:seltzer:hash)] was based on Litwin's Extensible Linear Hashing research. It boasted a clever scheme allowing a constant time mapping between hash values and page addresses, as well as the ability to handle large data—items larger than the underlying hash bucket or filesystem page size, typically four to eight kilobytes.

Berkeley DB项目开始于一个适度的目标——用一个新的、改进的、可同时支持内存和磁盘操作的哈希实现来替代内存哈希软件包hsearch和磁盘哈希软件包dbm/ndbm，以及允许不带专有许可的自由分发。Margo Seltzer写的哈希库 [[SY91](http://www.aosabook.org/en/bib1.html#bib:seltzer:hash)] 基于Litwin的可扩展线性哈希研究成果。它宣称采用了一种聪明的方法来达到哈希值和页面地址之间的常量时间映射，以及处理较大数据的能力——大于底层的哈希桶或文件系统页大小的项，通常是4到8KB。

If hash tables were good, then Btrees and hash tables would be better. Mike Olson, also a graduate student at the University of California, Berkeley, had written a number of Btree implementations, and agreed to write one more. The three of us transformed Margo's hash software and Mike's Btree software into an access-method-agnostic API, where applications reference hash tables or Btrees via database handles that had handle methods to read and modify data.

如果哈希表很好，那么B树加上哈希表将会更好。Mike Olson，也是加州大学伯克利分校的研究生，曾写过一些B树的实现，同意再写一个。我们三个人把Margo的哈希软件和Mike的B树软件转换成了一套和存取方法无关的API，应用程序通过数据库句柄来引用哈希表或B树，句柄带有读取或修改数据的处理方法。

Building on these two access methods, Mike Olson and Margo Seltzer wrote a research paper ([[SO92](http://www.aosabook.org/en/bib1.html#bib:seltzer:libtp)]) describing LIBTP, a programmatic transactional library that ran in an application's address space.

基于这两种存取方法，Mike Olson和Marge Seltzer写了一篇关于LIBTP（一个运行于应用程序地址空间的可编程事务函数库）的研究论文（([[SO92](http://www.aosabook.org/en/bib1.html#bib:seltzer:libtp)]）。

The hash and Btree libraries were incorporated into the final 4BSD releases, under the name Berkeley DB 1.85. Technically, the Btree access method implements a B+link tree, however, we will use the term Btree for the rest of this chapter, as that is what the access method is called. Berkeley DB 1.85's structure and APIs will likely be familiar to anyone who has used any Linux or BSD-based system.

这套哈希和B树函数库以Berkeley DB 1.85的名称被集成到了最终的4BSD发行版中。从技术上看，该B树存取方法实现的是B+ link树，不过在本章的后续部分我们将采用B树一词，因为它是存取方法的名称。Berkeley DB 1.85的结构和API对用过Linux或BSD衍生系统的人而言很可能比较熟悉。

The Berkeley DB 1.85 library was quiescent for a few years, until 1996 when Netscape contracted with Margo Seltzer and Keith Bostic to build out the full transactional design described in the LIBTP paper and create a production-quality version of the software. This effort produced the first transactional version of Berkeley DB, version 2.0.

Berkeley DB 1.85沉寂了一些年，直到1996年Netscape与Margo Seltzer和Keith Bostic签约来实现LIBTP论文中描述的全部事务设计并且实现一个生产质量级的版本。这项工作产生了Berkeley DB的第一个事务性版本，版本2.0。

The subsequent history of Berkeley DB is a simpler and more traditional timeline: Berkeley DB 2.0 (1997) introduced transactions to Berkeley DB; Berkeley DB 3.0 (1999) was a re-designed version, adding further levels of abstraction and indirection to accommodate growing functionality. Berkeley DB 4.0 (2001) introduced replication and high availability, and Oracle Berkeley DB 5.0 (2010) added SQL support.

Berkeley DB的后续历史就是一个更简单、传统的大事年表了：Berkeley DB 2.0（1997）引入了事务；Berkeley DB 3.0 （1999）是一个重新设计的版本，增加了更多级别的抽象和间接性以支持不断增长的功能；Berkeley DB 4.0 （2001）引入了复制和高可用；Oracle Berkeley DB 5.0 （2010）增加了SQL支持。

At the time of writing, Berkeley DB is the most widely used database toolkit in the world, with hundreds of millions of deployed copies running in everything from routers and browsers to mailers and operating systems. Although more than twenty years old, the Berkeley DB tool-based and object-oriented approach has allowed it to incrementally improve and re-invent itself to match the requirements of the software using it.

在写作本文的时候，Berkeley DB 是世界上使用最广泛的数据库工具集，有几亿份部署的拷贝运行在从路由器、浏览器、邮件系统到操作系统的各种系统中。虽然已经有超过20年的历史了，Berkeley DB 基于工具和面向对象的设计方法使得它可以增量改进和重构以满足使用它的软件的需求。

***Design Lesson 1 设计教训1***

It is vital for any complex software package's testing and maintenance that the software be designed and built as a cooperating set of modules with well-defined API boundaries. The boundaries can (and should!) shift as needs dictate, but they always need to be there. The existence of those boundaries prevents the software from becoming an unmaintainable pile of spaghetti. Butler Lampson once said that all problems in computer science can be solved by another level of indirection. More to the point, when asked what it meant for something to be object-oriented, Lampson said it meant being able to have multiple implementations behind an API. The Berkeley DB design and implementation embody this approach of permitting multiple implementations behind a common interface, providing an object-oriented look and feel, even though the library is written in C.

对任何复杂的软件包的测试和维护来说，将其设计和构建成带有良好定义的API边界的、一组互相协作的模块至关重要。在有需求时，这些边界能够（而且必须！）移动，但是边界总得存在。这些边界的存在可以防止软件变成一堆不可维护的意大利面条。Butler Lampson曾说过，计算机科学中的所有问题都可以通过添加一个间接层来解决。更确切的是，当被问及面向对象的东西是什么意思时，Lampson说这意味着能够在一套API之后有多种实现。Berkeley DB的设计和实现体现了这种同一套接口之后允许多种实现的方法，提供了面向对象的观感，虽然函数库是用C实现的。

***4.2. Architectural Overview 体系结构概述***

In this section, we'll review the Berkeley DB library's architecture, beginning with LIBTP, and highlight key aspects of its evolution.

本节我们将从LIBTP开始回顾Berkeley DB的体系结构，强调它演进中的关键问题。

[Figure 4.1](http://www.aosabook.org/en/bdb.html#fig.bdb.libtp), which is taken from Seltzer and Olson's original paper, illustrates the original LIBTP architecture, while [Figure 4.2](http://www.aosabook.org/en/bdb.html#fig.bdb.bdb20) presents the Berkeley DB 2.0 designed architecture.

图4.1摘自Seltzer和Olson的原始论文，说明了原先的LIBTP体系结构；而图4.2则展现了Berkeley DB 2.0设计时的体系结构。

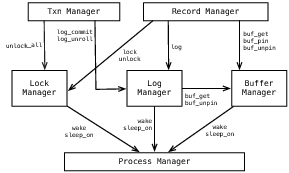


Figure 4.1: Architecture of the LIBTP Prototype System 图4.1：LIBTP原型系统的体系结构

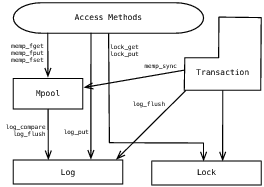


Figure 4.2: Intended Architecture for Berkeley DB-2.0. 图4.2：Berkeley DB 2.0预期的体系结构

The only significant difference between the LIBTP implementation and the Berkeley DB 2.0 design was the removal of the process manager. LIBTP required that each thread of control register itself with the library and then synchronized the individual threads/processes rather than providing subsystem level synchronization. As is discussed in [Section 4.4](http://www.aosabook.org/en/bdb.html#sec.bdb.int), that original design might have served us better.

LIBTP实现和Berkeley DB 2.0设计之间唯一显著的区别是删除了进程管理器（process manager）。LIBTP要求每个线程注册到库中，然后同步各个线程/进程，而不是提供子系统级的同步。正如4.4节中讨论的那样，原先的设计可能更好。

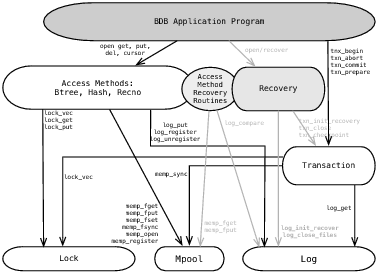


Figure 4.3: Actual Berkeley DB 2.0.6 Architecture. 图4.3：实际的Berkeley DB 2.0.6体系结构

The difference between the design and the actual released db-2.0.6 architecture, shown in [Figure 4.3](http://www.aosabook.org/en/bdb.html#fig.bdb.bdbact), illustrates the reality of implementing a robust recovery manager. The recovery subsystem is shown in gray. Recovery includes both the driver infrastructure, depicted in the recovery box, as well as a set of recovery redo and undo routines that recover the operations performed by the access methods. These are represented by the circle labelled "access method recovery routines." There is a consistent design to how recovery is handled in Berkeley DB 2.0 as opposed to hand-coded logging and recovery routines in LIBTP particular to specific access methods. This general purpose design also produces a much richer interface between the various modules.

设计和实际发布的Berkeley DB 2.0.6（见图4.3）在体系结构上的区别体现在后者实现了一个强壮的恢复管理器（recovery manager）。恢复子系统在图中用灰色表示。恢复既包括用recovery框表示的驱动结构，也包括用于恢复存取方法所执行操作的重做（redo）和撤销（undo）例程的集合。这些在图中用“access method recovery routines”标注的椭圆形表示。与LIBTP中针对特定存取方法编写日志和恢复例程不同，Berkeley DB 2.0中对恢复的处理是一种一致的设计。这个通用的设计也产生了不同模块间更丰富的接口。

[Figure 4.4](http://www.aosabook.org/en/bdb.html#fig.bdb.bdb50) illustrates the Berkeley DB-5.0.21 architecture. The numbers in the diagram reference the APIs listed in the table in [Table 4.1](http://www.aosabook.org/en/bdb.html#tbl.bdb.apitab). Although the original architecture is still visible, the current architecture shows its age with the addition of new modules, the decomposition of old modules (e.g., log has become log and dbreg), and a significant increase in the number of intermodule APIs).

图4.4展现了Berkeley DB 5.0.21的体系结构。图中的数字表示表4.1中列出的API。虽然仍可以看出原始的体系结构的样子，当前的体系结构体现了新模块的增加，旧模块的分解（例如log变成了log和dbreg），以及模块间API的显著增加。

Over a decade of evolution, dozens of commercial releases, and hundreds of new features later, we see that the architecture is significantly more complex than its ancestors. The key things to note are: First, replication adds an entirely new layer to the system, but it does so cleanly, interacting with the rest of the system via the same APIs as does the historical code. Second, the log module is split into log and dbreg (database registration). This is discussed in more detail in [Section 4.8](http://www.aosabook.org/en/bdb.html#sec.bdb.log). Third, we have placed all inter-module calls into a namespace identified with leading underscores, so that applications won't collide with our function names. We discuss this further in Design Lesson 6.

经过十年多的演进，几十个商业发布，以及几百个新特性的增加之后，我们看到体系结构明显比以前更复杂了。值得注意的关键点是：首先，复制（replication）在系统中增加了全新的一层，不过做得很清晰，就像前期的代码一样通过同样的API与系统的其他部分交互。其次，log模块被分成了log和dbreg（database registration）。在4.8节对此有更详细的描述。第三，我们把所有模块间的调用放到了一个以下划线打头的命名空间内，这样应用软件就不会与我们的函数名冲突了。我们在设计教训6中对此进一步讨论。

Fourth, the logging subsystem's API is now cursor based (there is no log\_get API; it is replaced by the log\_cursor API). Historically, Berkeley DB never had more than one thread of control reading or writing the log at any instant in time, so the library had a single notion of the current seek pointer in the log. This was never a good abstraction, but with replication it became unworkable. Just as the application API supports iteration using cursors, the log now supports iteration using cursors. Fifth, the fileop module inside of the access methods provides support for transactionally protected database create, delete, and rename operations. It took us multiple attempts to make the implementation palatable (it is still not as clean as we would like), and after reworking it numerous time, we pulled it out into its own module.

第四，日志子系统的API现在是基于游标的了（API log\_get不复存在，代之以API log\_cursor）。过去，Berkeley DB中在同一时刻读写日志的线程从来就没有多于一个，因此函数库中只有一个日志的当前扫描指针。这从来都不是一个好的抽象（但还可以工作），但有了复制之后它变得不可用了。就像应用层API用游标实现循环一样，日志现在也通过游标来支持循环了。第五，存取方法中的fileop模块提供了事务保护的数据库创建、删除和重命名操作。我们尝试了多次以使得实现使人满意（它仍然不是我们期望的那样清晰），在许多次改造之后，我们把它抽成一个独立的模块。

***Design Lesson 2 设计教训2***

A software design is simply one of several ways to force yourself to think through the entire problem before attempting to solve it. Skilled programmers use different techniques to this end: some write a first version and throw it away, some write extensive manual pages or design documents, others fill out a code template where every requirement is identified and assigned to a specific function or comment. For example, in Berkeley DB, we created a complete set of Unix-style manual pages for the access methods and underlying components before writing any code. Regardless of the technique used, it's difficult to think clearly about program architecture after code debugging begins, not to mention that large architectural changes often waste previous debugging effort. Software architecture requires a different mind set from debugging code, and the architecture you have when you begin debugging is usually the architecture you'll deliver in that release.

软件设计绝对是迫使你自己在试图解决问题前通盘考虑整个问题的几种方法之一。有经验的程序员采用不同的技术来达到这个目的：有些先写第一版然后扔掉，有些写出大量的手册或设计文档，其他的则设计出代码模板并识别出每个需求，分派到一个具体的函数或一段注释。例如，在Berkeley DB中，我们在写代码之前为存取方法和底层模块创建了一份完整的Unix风格的手册。不管采用的具体技术如何，在代码调试开始后都很难想清楚程序的体系结构，更不要说大的体系结构变化通常会浪费前期的调试努力。软件体系结构设计需要一种与代码调试不同的思维方式，当你开始调试时的软件体系结构通常就是你在该版本中将会交付的结构。

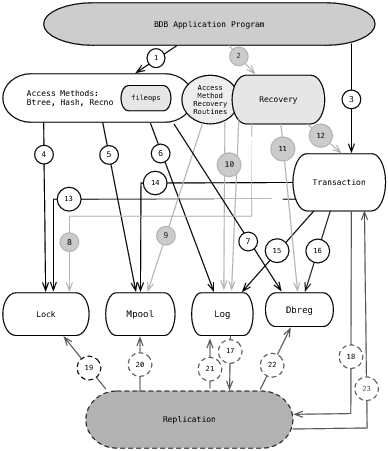


Figure 4.4: Berkeley DB-5.0.21 Architecture 图4.4：Berkeley DB 5.0.21的体系结构

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Application APIs** | | | | |
|  | | | | |
| **1. DBP handle operations** | | **2. DB\_ENV Recovery** | | **3. Transaction APIs** |
| open | | open(… DB\_RECOVER …) | | DB\_ENV->txn\_begin |
| get | |  | | DB\_TXN->abort |
| put | |  | | DB\_TXN->commit |
| del | |  | | DB\_TXN->prepare |
| cursor | |  | |  |
|  | | | | |
| **APIs Used by the Access Methods** | | | | |
|  | | | | |
| **4. Into Lock** | **5. Into Mpool** | **6. Into Log** | **7. Into Dbreg** |  |
| \_\_lock\_downgrade | \_\_memp\_nameop | \_\_log\_print\_record | \_\_dbreg\_setup |  |
| \_\_lock\_vec | \_\_memp\_fget |  | \_\_dbreg\_net\_id |  |
| \_\_lock\_get | \_\_memp\_fput |  | \_\_dbreg\_revoke |  |
| \_\_lock\_put | \_\_memp\_fset |  | \_\_dbreg\_teardown |  |
|  | \_\_memp\_fsync |  | \_\_dbreg\_close\_id |  |
|  | \_\_memp\_fopen |  | \_\_dbreg\_log\_id |  |
|  | \_\_memp\_fclose |  |  |  |
|  | \_\_memp\_ftruncate |  |  |  |
|  | \_\_memp\_extend\_freelist |  |  |  |
|  | | | | |
| **Recovery APIs** | | | | |
|  | | | | |
| **8. Into Lock** | **9. Into Mpool** | **10. Into Log** | **11. Into Dbreg** | **12. Into Txn** |
| \_\_lock\_getlocker | \_\_memp\_fget | \_\_log\_compare | \_\_dbreg\_close\_files | \_\_txn\_getckp |
| \_\_lock\_get\_list | \_\_memp\_fput | \_\_log\_open | \_\_dbreg\_mark\_restored | \_\_txn\_checkpoint |
|  | \_\_memp\_fset | \_\_log\_earliest | \_\_dbreg\_init\_recover | \_\_txn\_reset |
|  | \_\_memp\_nameop | \_\_log\_backup |  | \_\_txn\_recycle\_id |
|  |  | \_\_log\_cursor |  | \_\_txn\_findlastckp |
|  |  | \_\_log\_vtruncate |  | \_\_txn\_ckp\_read |
|  | | | | |
| **APIs Used by the Transaction Module** | | | | |
|  | | | | |
| **13. Into Lock** | **14. Into Mpool** | **15. Into Log** | **16. Into Dbreg** |  |
| \_\_lock\_vec | \_\_memp\_sync | \_\_log\_cursor | \_\_dbreg\_invalidate\_files |  |
| \_\_lock\_downgrade | \_\_memp\_nameop | \_\_log\_current\_lsn | \_\_dbreg\_close\_files |  |
|  |  |  | \_\_dbreg\_log\_files |  |
|  | | | | |
| **API Into the Replication System** | | | | |
|  | | | | |
|  |  | **17. From Log** |  | **18. From Txn** |
|  |  | \_\_rep\_send\_message |  | \_\_rep\_lease\_check |
|  |  | \_\_rep\_bulk\_message |  | \_\_rep\_txn\_applied |
|  |  |  |  | \_\_rep\_send\_message |
|  | | | | |
| **API From the Replication System** | | | | |
|  | | | | |
| **19. Into Lock** | **20. Into Mpool** | **21. Into Log** | **22. Into Dbreg** | **23. Into Txn** |
| \_\_lock\_vec | \_\_memp\_fclose | \_\_log\_get\_stable\_lsn | \_\_dbreg\_mark\_restored | \_\_txn\_recycle\_id |
| \_\_lock\_get | \_\_memp\_fget | \_\_log\_cursor | \_\_dbreg\_invalidate\_files | \_\_txn\_begin |
| \_\_lock\_id | \_\_memp\_fput | \_\_log\_newfile | \_\_dbreg\_close\_files | \_\_txn\_recover |
|  | \_\_memp\_fsync | \_\_log\_flush |  | \_\_txn\_getckp |
|  |  | \_\_log\_rep\_put |  | \_\_txn\_updateckp |
|  |  | \_\_log\_zero |  |  |
|  |  | \_\_log\_vtruncate |  |  |

Table 4.1: Berkeley DB 5.0.21 APIs 表4.1：Berkeley DB 5.0.21的API

Why architect the transactional library out of components rather than tune it to a single anticipated use? There are three answers to this question. First, it forces a more disciplined design. Second, without strong boundaries in the code, complex software packages inevitably degenerate into unmaintainable piles of glop. Third, you can never anticipate all the ways customers will use your software; if you empower users by giving them access to software components, they will use them in ways you never considered.

为什么把事务函数库设计成多个模块而不是为单一用途优化？针对这个问题有三个答案。首先，它促使一个更严谨的设计。其次，代码中若没有明显的边界，复杂的软件包必然会恶化成为一堆不可维护的东西。第三，你不可能预见用户使用你的软件的所有方式，如果你授权用户访问软件的内部模块，他们将会用你从未想到过的方式来使用这些模块。

In subsequent sections we'll consider each component of Berkeley DB, understand what it does and how it fits into the larger picture.

在随后的章节中，我们会讨论Berkeley DB中的每个组件，理解它做了什么以及它在整个系统中的位置。

***4.3. The Access Methods: Btree, Hash, Recno, Queue 存取方法：B树、哈希、记录号和队列***

The Berkeley DB access methods provide both keyed lookup of, and iteration over, variable and fixed-length byte strings. Btree and Hash support variable-length key/value pairs. Recno and Queue support record-number/value pairs (where Recno supports variable-length values and Queue supports only fixed-length values).

Berkeley DB的存取方法同时提供了基于变长和定长字节串的按键值查找和循环。B树和哈希支持变长的键/值对。记录号和队列支持记录号/值对（其中记录号支持变长值而队列仅支持定长值）。

Notes：Btree、Hash、Recno、Queue在这里属于专用名词，保留英文似乎更好。

The main difference between Btree and Hash access methods is that Btree offers locality of reference for keys, while Hash does not. This implies that Btree is the right access method for almost all data sets; however, the Hash access method is appropriate for data sets so large that not even the Btree indexing structures fit into memory. At that point, it's better to use the memory for data than for indexing structures. This trade-off made a lot more sense in 1990 when main memory was typically much smaller than today.

B树和哈希存取方法之间的主要区别在于B树提供了键值引用的局部性，而哈希则没有。这意味着对几乎所有的数据集B树都是合适的存取方法，而哈希存取方法则适合于大到连B树索引都在内存中放不下的数据集。此时，把内存用来存放数据比存放索引要更好。1990年那时的内存比今天要小很多，这种权衡显得更有道理。

The difference between Recno and Queue is that Queue supports record-level locking, at the cost of requiring fixed-length values. Recno supports variable-length objects, but like Btree and Hash, supports only page-level locking.

记录号和队列之间的差别在于队列以只支持定长值为代价来支持记录级锁定；记录号支持变长对象，但和B树以及哈希一样，仅支持页级锁定。

We originally designed Berkeley DB such that the CRUD functionality (create, read, update and delete) was key-based and the primary interface for applications. We subsequently added cursors to support iteration. That ordering led to the confusing and wasteful case of largely duplicated code paths inside the library. Over time, this became unmaintainable and we converted all keyed operations to cursor operations (keyed operations now allocate a cached cursor, perform the operation, and return the cursor to the cursor pool). This is an application of one of the endlessly-repeated rules of software development: don't optimize a code path in any way that detracts from clarity and simplicity until you know that it's necessary to do so.

我们最初把Berkeley DB设计成CRUD功能（创建、读取、更新和删除）是基于键的，而且是给应用的主要接口。后来我们增加了游标以支持循环。这个需求导致了函数库中大量的重复代码，造成了混乱和资源浪费。随着时间的推移，这变得不可维护，我们把所有基于键的操作都转换成了游标操作（现在，基于键的操作会分配一个缓存的游标，执行操作，然后将游标返回到游标池）。这是软件开发中不断重复的规则之一的应用：在你知道必须去做之前，不要以任何方式优化一条减少清晰度和简洁性的代码路径。

***Design Lesson 3 设计教训3***

Software architecture does not age gracefully. Software architecture degrades in direct proportion to the number of changes made to the software: bug fixes corrode the layering and new features stress design. Deciding when the software architecture has degraded sufficiently that you should re-design or re-write a module is a hard decision. On one hand, as the architecture degrades, maintenance and development become more difficult and at the end of that path is a legacy piece of software maintainable only by having an army of brute-force testers for every release, because nobody understands how the software works inside. On the other hand, users will bitterly complain over the instability and incompatibilities that result from fundamental changes. As a software architect, your only guarantee is that someone will be angry with you no matter which path you choose.

软件体系结构不会优雅地老化。软件体系结构的退化与软件本身的改动数量成正比：缺陷修复会腐蚀软件的层次，新特性会使设计产生应力。确定什么时候软件体系结构退化到该重新设计或重写一个模块是一个很难的决定。一方面，在设计退化时，维护和开发变得更困难，最终变成一个老化的软件。它的每次发布只能靠一群暴力测试者来维持。因为没有人知道该软件内部是怎么工作的。另一方面，用户会强烈抱怨根本性改动带来的不稳定和不兼容。作为一个软件架构师，你唯一的保证是无论选择那条路，总有人对你不满。

We omit detailed discussions of the Berkeley DB access method internals; they implement fairly well-known Btree and hashing algorithms (Recno is a layer on top of the Btree code, and Queue is a file block lookup function, albeit complicated by the addition of record-level locking).

我们略去了对Berkeley DB存取方法内部的详细讨论。他们实现了众所周知的B树和哈希算法（记录号是B树代码之上的一层；队列是一个文件块查找功能，尽管它被记录级锁定弄复杂了。）

***4.4. The Library Interface Layer 函数库的接口层***

Over time, as we added additional functionality, we discovered that both applications and internal code needed the same top-level functionality (for example, a table join operation uses multiple cursors to iterate over the rows, just as an application might use a cursor to iterate over those same rows).

随着时间的推移，我们增加了更多的功能，发现应用程序和内部代码都需要相同的上层功能（例如内部的表连接操作要用到多个游标来遍历行，应用程序也会用游标来遍历同样这些行。）

***Design Lesson 4 设计教训4***

It doesn't matter how you name your variables, methods, functions, or what comments or code style you use; that is, there are a large number of formats and styles that are "good enough." What does matter, and matters very much, is that naming and style be consistent. Skilled programmers derive a tremendous amount of information from code format and object naming. You should view naming and style inconsistencies as some programmers investing time and effort to lie to the other programmers, and vice versa. Failing to follow house coding conventions is a firing offense.

你怎么命名变量、方法和函数，采用什么注释或代码风格并不重要；也就是说有大量的格式和风格“足够好”。重要和非常重要的是命名和风格保持一致。有经验的程序员从代码格式和对象命名中得到大量信息。你应当将命名和风格的不一致视为某些程序员将时间和精力花费来欺骗另外的程序员，反之亦然。不能遵循内部编码规范是一种该被解雇的行为。

For this reason, we decomposed the access method APIs into precisely defined layers. These layers of interface routines perform all of the necessary generic error checking, function-specific error checking, interface tracking, and other tasks such as automatic transaction management. When applications call into Berkeley DB, they call the first level of interface routines based on methods in the object handles. (For example, \_\_dbc\_put\_pp, is the interface call for the Berkeley DB cursor "put" method, to update a data item. The "\_pp" is the suffix we use to identify all functions that an application can call.)

正因如此，我们把存取方法的API分拆为准确定义的层次。这些接口例程层处理所有必要的通用错误检查，函数特有的错误检查，接口追踪以及其他如自动事务管理等任务。当应用程序调用进Berkeley DB时，它们调用的是基于对象句柄内的方法的第一层接口例程（例如Berkeley DB游标的“put”方法就是调用\_\_dbc\_put\_pp接口来更新数据项的）。我们用后缀“\_pp”来标识所有可以被应用程序调用的函数。

One of the Berkeley DB tasks performed in the interface layer is tracking what threads are running inside the Berkeley DB library. This is necessary because some internal Berkeley DB operations may be performed only when no threads are running inside the library. Berkeley DB tracks threads in the library by marking that a thread is executing inside the library at the beginning of every library API and clearing that flag when the API call returns. This entry/exit checking is always performed in the interface layer, as is a similar check to determine if the call is being performed in a replicated environment.

Berkeley DB的接口层处理的任务之一是追踪哪些线程正在Berkeley DB库内运行。这是必要的，因为有些内部的Berkeley DB操作只可以在库内没有线程运行时被执行。Berkeley DB通过在每个库API开始时标记线程在库内运行，在API调用返回时清除标记来追踪线程。这些进入/退出检查总是在接口层进行检查，与此类似的是检查调用是否在复制环境中执行。

The obvious question is "why not pass a thread identifier into the library, wouldn't that be easier?" The answer is yes, it would be a great deal easier, and we surely wish we'd done just that. But, that change would have modified every single Berkeley DB application, most of every application's calls into Berkeley DB, and in many cases would have required application re-structuring.

很明显的一个问题是“为什么不传递一个线程标识符到函数库，这难道不是更简单吗？”答案是肯定的，那将容易很多，我们当然希望已经那么做了。可是，这种变化将导致每个Berkeley DB应用程序，以及每个应用程序中对Berkeley DB的大部分调用，这在大部分情况下需要应用程序的重构。

***Design Lesson 5 设计教训5***

Software architects must choose their upgrade battles carefully: users will accept minor changes to upgrade to new releases (if you guarantee compile-time errors, that is, obvious failures until the upgrade is complete; upgrade changes should never fail in subtle ways). But to make truly fundamental changes, you must admit it's a new code base and requires a port of your user base. Obviously, new code bases and application ports are not cheap in time or resources, but neither is angering your user base by telling them a huge overhaul is really a minor upgrade.

软件架构师必须慎重选择升级路径：用户会接受小的改动来升级到新的版本（如果你保证升级期间只出现编译时错误也就是明显的错误；升级的变化绝不应该导致难以理解的失败。）但是要做出真正根本性的变化，你必须承认这是一个新的基础代码，所以需要现有用户的移植。显然，新的基础代码和应用移植在时间或资源上算都不便宜，但是二者都不会像告诉他们一个大改动实际是一次小升级那样惹恼你的用户群。

Another task performed in the interface layer is transaction generation. The Berkeley DB library supports a mode where every operation takes place in an automatically generated transaction (this saves the application having to create and commit its own explicit transactions). Supporting this mode requires that every time an application calls through the API without specifying its own transaction, a transaction is automatically created.

接口层负责的另一个任务是事务的产生。Berkeley DB支持一种每个操作都隐含一个事务的模式（这可以省去应用程序显式创建和提交事务的麻烦）。支持这种模式需要在应用程序未指定自己的事务调用API时，自动为其创建一个事务。

Finally, all Berkeley DB APIs require argument checking. In Berkeley DB there are two flavors of error checking—generic checks to determine if our database has been corrupted during a previous operation or if we are in the midst of a replication state change (for example, changing which replica allows writes). There are also checks specific to an API: correct flag usage, correct parameter usage, correct option combinations, and any other type of error we can check before actually performing the requested operation.

最后，所有的Berkeley DB API都需要参数检查。在Berkeley DB中有两种类型的错误检查——判断数据库是否在前一个操作中被破坏了的通用性检查，以及我们是否正在一个复制状态变化的中间（例如，改变哪个副本以允许写入）。也有针对具体API的检查：标记的正确使用，参数的正确使用，选项的正确组合，以及其他可以在真正执行请求的操作前检查的错误。

This API-specific checking is all encapsulated in functions suffixed with \_arg. Thus, the error checking specific to the cursor put method is located in the function \_\_dbc\_put\_arg, which is called by the \_\_dbc\_put\_pp function.

API相关的检查都被封装在以“\_arg”为后缀的函数中。因此，与游标的put方法相关的错误检查就位于\_\_dbc\_put\_arg中，它被函数\_\_dbc\_put\_pp调用。

Finally, when all the argument verification and transaction generation is complete, we call the worker method that actually performs the operation (in our example, it would be \_\_dbc\_put), which is the same function we use when calling the cursor put functionality internally.

最后，当所有参数检验和事务产生完成时，我们调用真正执行操作的辅助方法（在上述例子中是\_\_dbc\_put），这也是我们内部调用游标put功能时用的函数。

This decomposition evolved during a period of intense activity, when we were determining precisely what actions we needed to take when working in replicated environments. After iterating over the code base some non-trivial number of times, we pulled apart all this preamble checking to make it easier to change the next time we identified a problem with it.

这种模块拆分在一段开发密集期间逐渐形成，那时我们正在决策需要采取哪些行动以支持复制环境。在基础代码中迭代开发不少次后，我们把前面所说的所有检查都抽出来以使得以后发现问题时更容易修改。

***4.5. The Underlying Components 底层模块***

There are four components underlying the access methods: a buffer manager, a lock manager, a log manager and a transaction manager. We'll discuss each of them separately, but they all have some common architectural features.

在存取方法之下有四个模块：缓冲区管理器、锁管理器、日志管理器和事务管理器。我们将分别讨论每个模块，不过它们有一些共同的体系结构特性。

First, all of the subsystems have their own APIs, and initially each subsystem had its own object handle with all methods for that subsystem based on the handle. For example, you could use Berkeley DB's lock manager to handle your own locks or to write your own remote lock manager, or you could use Berkeley DB's buffer manager to handle your own file pages in shared memory. Over time, the subsystem-specific handles were removed from the API in order to simplify Berkeley DB applications. Although the subsystems are still individual components that can be used independently of the other subsystems, they now share a common object handle, the DB\_ENV "environment" handle. This architectural feature enforces layering and generalization. Even though the layer moves from time-to-time, and there are still a few places where one subsystem reaches across into another subsystem, it is good discipline for programmers to think about the parts of the system as separate software products in their own right.

首先，所有的这些子系统都有自己的API，而且最初每个子系统都有自己的对象句柄，子系统的所有方法都基于该句柄。例如，你可以用Berkeley DB的锁管理器来处理你自己的锁或者写自己的远程锁管理器。你也可以用Berkeley DB的内存管理器来处理自己的共享内存中的文件页。随着时间的推移，这些子系统特性的句柄被从API中删除了以简化Berkeley DB应用程序。虽然这些子系统仍然是可以被独立于其他子系统使用的独立模块，它们现在共享一个通用的对象句柄，也就是DB\_ENV“环境”句柄。这个体系结构的特性强化了分层和通用性。虽然层不时在移动，而且还有些地方一个子系统跨越到另一个子系统，让程序员把系统的不同部分理解为各自独立的软件产品是一个不错的原则。

Second, all of the subsystems (in fact, all Berkeley DB functions) return error codes up the call stack. As a library, Berkeley DB cannot step on the application's name space by declaring global variables, not to mention that forcing errors to return in a single path through the call stack enforces good programmer discipline.

其次，所有的这些子系统（实际上，所有的Berkeley DB函数）都给上层返回错误码。作为一个函数库，Berkeley DB不能通过定义全局变量侵入应用程序的名字空间。更何况强制错误从调用栈通过单一路径返回强化了好的程序员纪律。

***Design Lesson 6 设计经验6***

In library design, respect for the namespace is vital. Programmers who use your library should not need to memorize dozens of reserved names for functions, constants, structures, and global variables to avoid naming collisions between an application and the library.

在函数库的设计中，重视名字空间是至关重要的。用你的函数库的程序员应该不需要去记住几十个函数、常量、结构、全局变量的保留名字以避免应用和函数库的命名冲突。

Finally, all of the subsystems support shared memory. Because Berkeley DB supports sharing databases between multiple running processes, all interesting data structures have to live in shared memory. The most significant implication of this choice is that in-memory data structures must use base address and offset pairs instead of pointers in order for pointer-based data structures to work in the context of multiple processes. In other words, instead of indirecting through a pointer, the Berkeley DB library must create a pointer from a base address (the address at which the shared memory segment is mapped into memory) plus an offset (the offset of a particular data structure in that mapped-in segment). To support this feature, we wrote a version of the Berkeley Software Distribution queue package that implemented a wide variety of linked lists.

最后，所有这些子系统都支持共享内存。因为Berkeley DB支持在多个运行的进程之间共享数据库，所有共享数据结构都必须放在共享内存中。这个选择的最明显的结果是内存数据结构都必须采用一对基地址和偏移量而不是指针，以使得基于指针的数据结构都可以在多进程的环境下工作。换句话说，不通过指针做间接转换，Berkeley DB函数库必须通过基地址（共享内存段被映射到进程中的内存地址）加上一个偏移量（给定数据结构在映射的内存段中的偏移位置）来创建指针。为了支持这个特性，我们写了一个BSD版本的queue软件包，它实现了各种各样的链表。

***Design Lesson 7 设计教训7***

Before we wrote a shared-memory linked-list package, Berkeley DB engineers hand-coded a variety of different data structures in shared memory, and these implementations were fragile and difficult to debug. The shared-memory list package, modeled after the BSD list package (queue.h), replaced all of those efforts. Once it was debugged, we never had to debug another shared memory linked-list problem. This illustrates three important design principles: First, if you have functionality that appears more than once, write the shared functions and use them, because the mere existence of two copies of any specific functionality in your code guarantees that one of them is incorrectly implemented. Second, when you develop a set of general purpose routines, write a test suite for the set of routines, so you can debug them in isolation. Third, the harder code is to write, the more important for it to be separately written and maintained; it's almost impossible to keep surrounding code from infecting and corroding a piece of code.

在我们写共享内存的链表软件包之前，Berkeley DB的工程师们手工编写了共享内存中的各式不同的数据结构，而且这些实现容易出错和很难调试。共享内存链表软件包，仿照BSD链表软件包（queue.h）实现，代替了所有这些努力。在它一旦调试通过后，我们再也不需要去调试共享内存链表问题了。这体现了三个重要的设计原则：第一，如果一个功能出现了多次，那就写出共享的函数并使用它们，因为对于任何特定功能而言，两份拷贝的存在一定说明其中一份实现得不正确。其次，当你开发一系列通用的例程时，给这些例程写一个测试集，这样你就可以分开调试它们。第三，代码越难以书写，单独书写并维护它就越重要。因为基本上不可能防止外围代码感染和侵蚀一份代码。

***4.6. The Buffer Manager: Mpool 缓冲区管理器：Mpool***

The Berkeley DB Mpool subsystem is an in-memory buffer pool of file pages, which hides the fact that main memory is a limited resource, requiring the library to move database pages to and from disk when handling databases larger than memory. Caching database pages in memory was what enabled the original hash library to significantly out-perform the historic hsearch and ndbm implementations.

Berkeley DB的Mpool子系统是文件页面的内存缓冲池，它隐藏了这样一个事实：内存是一种有限资源，当处理超过内存大小的数据库时，需要函数库在磁盘和内存间来回移动数据库页。将数据库页缓存在内存中使得原先的哈希库大大优于先前的hsearch和hdbm实现。

Although the Berkeley DB Btree access method is a fairly traditional B+tree implementation, pointers between tree nodes are represented as page numbers, not actual in-memory pointers, because the library's implementation uses the on-disk format as its in-memory format as well. The advantage of this representation is that a page can be flushed from the cache without format conversion; the disadvantage is that traversing an index structures requires (costlier) repeated buffer pool lookups rather than (cheaper) memory indirections.

虽然Berkeley DB的B树存取方法是一个相当传统的B+树实现，树节点之间的指针用页面号而不是内存指针表示，因为函数也把磁盘页格式用作内存页格式。这种表示的优势在于页面可以不需要格式转换就能被从缓存刷出到磁盘，劣势在于遍历索引结构时需要（代价稍高的）重复的缓冲池查找而不是（代价稍低的）内存操作。

There are other performance implications that result from the underlying assumption that the in-memory representation of Berkeley DB indices is really a cache for on-disk persistent data. For example, whenever Berkeley DB accesses a cached page, it first pins the page in memory. This pin prevents any other threads or processes from evicting it from the buffer pool. Even if an index structure fits entirely in the cache and need never be flushed to disk, Berkeley DB still acquires and releases these pins on every access, because the underlying model provided by Mpool is that of a cache, not persistent storage.

底层假设Berkeley DB索引的内存表示实际上是磁盘上持久数据的缓存，这还有其他的一些对性能的影响。例如，每当Berkeley DB访问一个缓存的页面时，首先要pin住内存中的页面。Pin操作防止任何其他的线程或进程将该页从内存池中换出。即便整个索引结构都可以在缓冲中放下，并且从不需要被刷新到磁盘，Berkeley DB仍然在每个操作时要获取和释放这些pin，因为Mpool底层的模型是一个缓存而不是一个持久存储。

***4.6.1. The Mpool File Abstraction Mpool的文件抽象***

Mpool assumes it sits atop a filesystem, exporting the file abstraction through the API. For example, DB\_MPOOLFILE handles represent an on-disk file, providing methods to get/put pages to/from the file. While Berkeley DB supports temporary and purely in-memory databases, these too are referenced by DB\_MPOOLFILE handles because of the underlying Mpool abstractions. The get and put methods are the primary Mpool APIs: get ensures a page is present in the cache, acquires a pin on the page and returns a pointer to the page. When the library is done with the page, the put call unpins the page, releasing it for eviction. Early versions of Berkeley DB did not differentiate between pinning a page for read access versus pinning a page for write access. However, in order to increase concurrency, we extended the Mpool API to allow callers to indicate their intention to update a page. This ability to distinguish read access from write access was essential to implement multi-version concurrency control. A page pinned for reading that happens to be dirty can be written to disk, while a page pinned for writing cannot, since it may be in an inconsistent state at any instant.

Mpool假设它位于文件系统之上，通过其API暴露文件抽象。例如，DB\_MPOOLFILE句柄表示一个磁盘文件，提供了从文件中获取页面和写页面到文件的方法。虽然Berkeley DB也支持临时的和纯粹的内存数据库，这二者也是通过DB\_MPOOLFILE句柄引用的，因为底层都是Mpool抽象层。Get和put方法是主要的Mpool API：get确保页面在缓存中，获得页面上的一个pin并返回指向页面的指针。当函数库用完页面时，put调用unpin页面并允许页面被换出。Berkeley DB的早期版本不区分读访问的pin页面和写访问的pin页面。然而，为了增加并发性，我们扩展了Mpool的API以允许调用者指示更新页面的意图。区分读访问和写访问的能力对多版本并发控制的实现至为重要。为读访问pin住的脏页面是可以被写入磁盘的，而为写访问pin住的脏页面就不能，因为后者可能在任何时刻都处于不一致的状态。

***4.6.2. Write-ahead Logging 先写日志***

Berkeley DB uses write-ahead-logging (WAL) as its transaction mechanism to make recovery after failure possible. The term write-ahead-logging defines a policy requiring log records describing any change be propagated to disk *before* the actual data updates they describe. Berkeley DB's use of WAL as its transaction mechanism has important implications for Mpool, and Mpool must balance its design point as a generic caching mechanism with its need to support the WAL protocol.

Berkeley DB采用先写日志（WAL）实现故障恢复的事务机制。术语先写日志定义了一个策略，要求任何修改所对应的日志记录都要先于它实际的数据更新被写到磁盘。Berkeley DB采用WAL作为其事务机制对Mpool有重要的影响，Mpool必须在通用的缓存机制以及支持WAL协议的需要之间找到设计的平衡点。

Berkeley DB writes log sequence numbers (LSNs) on all data pages to document the log record corresponding to the most recent update to a particular page. Enforcing WAL requires that before Mpool writes any page to disk, it must verify that the log record corresponding to the LSN on the page is safely on disk. The design challenge is how to provide this functionality without requiring that all clients of Mpool use a page format identical to that used by Berkeley DB. Mpool addresses this challenge by providing a collection of set (and get) methods to direct its behavior. The DB\_MPOOLFILE method set\_lsn\_offset provides a byte offset into a page, indicating where Mpool should look for an LSN to enforce WAL. If the method is never called, Mpool does not enforce the WAL protocol. Similarly, the set\_clearlen method tells Mpool how many bytes of a page represent metadata that should be explicitly cleared when a page is created in the cache. These APIs allow Mpool to provide the functionality necessary to support Berkeley DB's transactional requirements, without forcing all users of Mpool to do so.

Berkeley DB将日志顺序号（LSN）写到所有数据页上，以记录每个特定页的最近更新所对应的日志记录。实施WAL需要Mpool在写页面到磁盘前验证页面上的LSN对应的日志记录已经安全地记录到磁盘了。设计的挑战在于提供该功能而不要求所有Mpool的客户采用和Berkeley DB完全一致的页面格式。Mpool通过提供一系列的set（和get）方法指引其行为来解决这个挑战。DB\_MPOOLFILE的方法set\_lsn\_offset提供了页面内的字节偏移，告诉Mpool到哪儿去找LSN以实现WAL。如果这个方法从未被调过，Mpool就不实现WAL。类似的，set\_clearlen方法告诉Mpool页内有多少字节表示元数据，在缓存中创建一个页面前需要显式的清除掉这些字节。这些API允许Mpool提供了支持Berkeley DB事务所必要的功能，而不是迫使Mpool的所有用户去自己实现。

***Design Lesson 8 设计教训8***

Write-ahead logging is another example of providing encapsulation and layering, even when the functionality is never going to be useful to another piece of software: after all, how many programs care about LSNs in the cache? Regardless, the discipline is useful and makes the software easier to maintain, test, debug and extend.

先写日志是另一个提供封装和分层的例子，即使是这个特性不会对其他的软件有用：毕竟有多少程序会关心缓存中的LSN？不管怎样，这个原则是有用的，而且使得软件容易维护、测试、调试和扩展。

***4.7. The Lock Manager: Lock 锁管理器：Lock***

Like Mpool, the lock manager was designed as a general-purpose component: a hierarchical lock manager (see [[GLPT76](http://www.aosabook.org/en/bib1.html#bib:gray:lock)]), designed to support a hierarchy of objects that can be locked (such as individual data items), the page on which a data item lives, the file in which a data item lives, or even a collection of files. As we describe the features of the lock manager, we'll also explain how Berkeley DB uses them. However, as with Mpool, it's important to remember that other applications can use the lock manager in completely different ways, and that's OK—it was designed to be flexible and support many different uses.

像Mpool一样，锁管理器也被设计成一个通用模块：它被设计成支持对象层次的封锁（例如独立的数据项、数据项所在的页面、甚至是一组文件）的一个层次式锁管理器（参看[[GLPT76](http://www.aosabook.org/en/bib1.html#bib:gray:lock)]）。在描述锁管理器的特性时，也将同时解释Berkeley DB是怎么用它的。然而，就像Mpool一样，其他的应用程序可以用完全不同的方式使用锁管理器，不过那没问题——它被设计得很灵活并支持很多不同的用法。

The lock manager has three key abstractions: a "locker" that identifies on whose behalf a lock is being acquired, a "lock\_object" that identifies the item being locked, and a "conflict matrix".

锁管理器有三个关键的抽象：“封锁者”标识锁是代表谁获取的，“封锁对象”标识被锁定的项，以及一个“冲突矩阵”。

Lockers are 32-bit unsigned integers. Berkeley DB divides this 32-bit name space into transactional and non-transactional lockers (although that distinction is transparent to the lock manager). When Berkeley DB uses the lock manager, it assigns locker IDs in the range 0 to 0x7fffffff to non-transactional lockers and the range 0x80000000 to 0xffffffff to transactions. For example, when an application opens a database, Berkeley DB acquires a long-term read lock on that database to ensure no other thread of control removes or renames it while it is in-use. As this is a long-term lock, it does not belong to any transaction and the locker holding this lock is non-transactional.

封锁者是32位无符号整数。Berkeley DB把这个32位的名字空间划分为事务性封锁者和非事务性封锁者（虽然这种区分对锁管理器而言是透明的）。当Berkeley DB使用锁管理器时，它把范围从0到0x7fffffff之间的ID分给非事务性封锁者，把从0x80000000到0xffffffff的分给事务性封锁者。例如，当应用程序打开数据库时，Berkeley DB获取该数据库上的一个长的读锁以保证它在被使用时没有其他的线程删除或重命名它。因为这是一个长锁，所以它不属于任何一个事务，持有该锁的封锁者就是非事务性的。

Any application using the lock manager needs to assign locker ids, so the lock manager API provides both DB\_ENV->lock\_id and DB\_ENV->lock\_id\_free calls to allocate and deallocate lockers. So applications need not implement their own locker ID allocator, although they certainly can.

任何使用锁管理器的应用程序都需要分配封锁者ID，所以锁管理器的API同时提供了DB\_ENV->lock\_id和DB\_ENV->lock\_id\_free调用用以分配和释放封锁者。因此应用程序不需要实现自己的封锁者ID分配器，虽然他们也可以这么做。

***4.7.1. Lock Objects 锁对象***

Lock objects are arbitrarily long opaque byte-strings that represent the objects being locked. When two different lockers want to lock a particular object, they use the same opaque byte string to reference that object. That is, it is the application's responsibility to agree on conventions for describing objects in terms of opaque byte strings.

锁对象是表示被封锁对象的任意长度的不透明（opaque）字节串。当两个不同的封锁者试图锁住一个特定对象时，他们采用同样的不透明字节串来引用该对象。也就是说，应用程序负责定义描述对象的不透明字节串的约定。

For example, Berkeley DB uses a DB\_LOCK\_ILOCK structure to describe its database locks. This structure contains three fields: a file identifier, a page number, and a type.

例如，Berkeley DB采用一个DB\_LOCK\_ILOCK结构来描述其数据库锁。这个结构包含三个字段：文件标识符、页号和类型。

In almost all cases, Berkeley DB needs to describe only the particular file and page it wants to lock. Berkeley DB assigns a unique 32-bit number to each database at create time, writes it into the database's metadata page, and then uses it as the database's unique identifier in the Mpool, locking, and logging subsystems. This is the fileid to which we refer in the DB\_LOCK\_ILOCK structure. Not surprisingly, the page number indicates which page of the particular database we wish to lock. When we reference page locks, we set the type field of the structure to DB\_PAGE\_LOCK. However, we can also lock other types of objects as necessary. As mentioned earlier, we sometimes lock a database handle, which requires a DB\_HANDLE\_LOCK type. The DB\_RECORD\_LOCK type lets us perform record level locking in the queue access method, and the DB\_DATABASE\_LOCK type lets us lock an entire database.

在几乎所有情况下，Berkeley DB都只需要描述它想锁定的特定文件和页面。Berkeley DB在数据库创建时给每个库分配一个唯一的32位数字，并把它写到数据库的元数据页中。以后就在Mpool、封锁、日志子系统中将它用作数据库的唯一标识符。这就是我们在DB\_LOCK\_ILOCK结构中引用的fileid字段。不出所料，页面号表示我们想要锁定的特定数据库中的某个页。当我们引用页面锁时，我们将结构中的type字段设置为DB\_PAGE\_LOCK。然而，我们我们也可以在需要时锁定其他类型的对象。正如前面提到的，我们有时会锁住数据库句柄，它就需要DB\_HANDLE\_LOCK类型。DB\_RECORD\_LOCK类型使我们可以处理队列存取方法中的记录级锁定，而DB\_DATABASE\_LOCK类型则让我们锁定整个数据库。

***Design Lesson 9 设计教训9***

Berkeley DB's choice to use page-level locking was made for good reasons, but we've found that choice to be problematic at times. Page-level locking limits the concurrency of the application as one thread of control modifying a record on a database page will prevent other threads of control from modifying other records on the same page, while record-level locks permit such concurrency as long as the two threads of control are not modifying the same record. Page-level locking enhances stability as it limits the number of recovery paths that are possible (a page is always in one of a couple of states during recovery, as opposed to the infinite number of possible states a page might be in if multiple records are being added and deleted to a page). As Berkeley DB was intended for use as an embedded system where no database administrator would be available to fix things should there be corruption, we chose stability over increased concurrency.

Berkeley DB选择采用页面级别的锁定是有足够理由的，但是我们发现该选择有时也是有问题的。当一个线程在修改数据库页面中的一条记录时，页级锁定将不允许其他线程修改同一页面中的其他记录，这限制了应用程序的并发性。而只要两个线程不在修改同一个记录，记录级锁定就允许这样的并发。页级锁定增强了稳定性，因为它限制了可能的恢复路径（在恢复过程中，页面总是在几个状态之一，而不是在允许多个记录被同时在页内增加或删除时导致的无数的可能状态）。因为Berkeley DB是为嵌入式系统使用的，一旦有破坏，不会有数据库系统管理员来修复问题，我们选择了稳定性而不是更好的并发。

***4.7.2. The Conflict Matrix 冲突矩阵***

The last abstraction of the locking subsystem we'll discuss is the conflict matrix. A conflict matrix defines the different types of locks present in the system and how they interact. Let's call the entity holding a lock, the holder and the entity requesting a lock the requester, and let's also assume that the holder and requester have different locker ids. The conflict matrix is an array indexed by [requester][holder], where each entry contains a zero if there is no conflict, indicating that the requested lock can be granted, and a one if there is a conflict, indicating that the request cannot be granted.

我们将讨论的封锁子系统的最后一个抽象是冲突矩阵。冲突矩阵定义了系统中不同类型的锁以及它们之间的交互。让我们将持有锁的实体称为持有者，请求锁的称为请求者，并且假设持有者和请求者具有不同的封锁者ID。冲突矩阵就是一个以[requester][holder]为下标的数组，其中如果没有冲突的格子为0，表明请求的锁可以被授予，如果有冲突则为1，表明请求不能被授予。

The lock manager contains a default conflict matrix, which happens to be exactly what Berkeley DB needs, however, an application is free to design its own lock modes and conflict matrix to suit its own purposes. The only requirement on the conflict matrix is that it is square (it has the same number of rows and columns) and that the application use 0-based sequential integers to describe its lock modes (e.g., read, write, etc.). [Table 4.2](http://www.aosabook.org/en/bdb.html#tbl.bdb.two) shows the Berkeley DB conflict matrix.

锁管理器含有一个缺省的冲突矩阵，它碰巧正是Berkeley DB所需要的。然而，应用程序可以自由定义自己的封锁模式和冲突矩阵以满足它自己的需求。对冲突矩阵的唯一要求是它必须是方的（它有相同的行数和列数）并且应用程序用从0开始的整数描述其封锁模式（例如读、写等）。表4.2列出了Berkeley DB的冲突矩阵。

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Holder** | | | | | | | | |  |  |  |  |  |  |  |
| **Requester** | **No-Lock** | **Read** | **Write** | **Wait** | **iWrite** | **iRead** | **iRW** | **uRead** | **wasWrite** |  |  |  |  |  |  |  |
| **No-Lock** |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **Read** |  |  | ✓ |  | ✓ |  | ✓ |  | ✓ |  |  |  |  |  |  |  |
| **Write** |  | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |  |  |  |  |  |  |  |
| **Wait** |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **iWrite** |  | ✓ | ✓ |  |  |  |  | ✓ | ✓ |  |  |  |  |  |  |  |
| **iRead** |  |  | ✓ |  |  |  |  |  | ✓ |  |  |  |  |  |  |  |
| **iRW** |  | ✓ | ✓ |  |  |  |  | ✓ | ✓ |  |  |  |  |  |  |  |
| **uRead** |  |  | ✓ |  | ✓ |  | ✓ |  |  |  |  |  |  |  |  |  |
| **iwasWrite** |  | ✓ | ✓ |  | ✓ | ✓ | ✓ |  | ✓ |  |  |  |  |  |  |  |

Table 4.2: Read-Writer Conflict Matrix. 表4.2：读写冲突矩阵

***4.7.3. Supporting Hierarchical Locking对层次封锁的支持***

Before explaining the different lock modes in the Berkeley DB conflict matrix, let's talk about how the locking subsystem supports hierarchical locking. Hierarchical locking is the ability to lock different items within a containment hierarchy. For example, files contain pages, while pages contain individual elements. When modifying a single page element in a hierarchical locking system, we want to lock just that element; if we were modifying every element on the page, it would be more efficient to simply lock the page, and if we were modifying every page in a file, it would be best to lock the entire file. Additionally, hierarchical locking must understand the hierarchy of the containers because locking a page also says something about locking the file: you cannot modify the file that contains a page at the same time that pages in the file are being modified.

在解释Berkeley DB冲突矩阵中不同的封锁模式之前，让我们谈谈封锁子系统是怎么支持层次封锁的。层次封锁指的是一种锁定同一层次结构中不同项的能力。例如文件包含页面，而页面包含不同的元素。当在一个层次封锁系统中修改一个页面元素时，我们仅想锁住该元素；如果我们要更新页面中的每个元素，仅锁定页面将更有效，而如果我们要修改文件中的每个页面，最好的就是锁定整个文件。此外，层次封锁必须理解容器的层次，因为锁定一个页面也意味着在某种程度上锁定了文件，文件中有页面在被修改时，你不能修改页面所在的文件。

The question then is how to allow different lockers to lock at different hierarchical levels without chaos resulting. The answer lies in a construct called an intention lock. A locker acquires an intention lock on a container to indicate the intention to lock things within that container. So, obtaining a read-lock on a page implies obtaining an intention-to-read lock on the file. Similarly, to write a single page element, you must acquire an intention-to-write lock on both the page and the file. In the conflict matrix above, the iRead, iWrite, and iWR locks are all intention locks that indicate an intention to read, write or do both, respectively.

那么问题在于怎么允许不同的封锁者在不同层级进行封锁又不引起混乱。答案是一种叫做意向锁的结构。封锁者获取容器上的一个意向锁以说明将要锁定容器内的东西的意向。于是，获取页面上的读锁隐含着获取文件上的一个意向读锁。类似的，要写页面中的一个元素，你必须同时获取页面和文件上的意向写锁。在上面的冲突矩阵中，iRead、iWrite和iWR锁都是意向锁，它们分别表示读的意向、写的意向和同时读写的意向。

Therefore, when performing hierarchical locking, rather than requesting a single lock on something, it is necessary to request potentially many locks: the lock on the actual entity as well as intention locks on any containing entities. This need leads to the Berkeley DB DB\_ENV->lock\_vec interface, which takes an array of lock requests and grants them (or rejects them), atomically.

因此，在处理层次封锁时，不是在某个东西上请求单一的一个锁，可能有必要请求很多锁：最终要操作的实体上的锁以及所有包含该实体的实体的意向锁。这个需求引入了Berkeley DB中的DB\_ENV->lock\_vec接口，它接受一个锁请求的数组然后原子性的授予（或拒绝）。

Although Berkeley DB doesn't use hierarchical locking internally, it takes advantage of the ability to specify different conflict matrices, and the ability to specify multiple lock requests at once. We use the default conflict matrix when providing transactional support, but a different conflict matrix to provide simple concurrent access without transaction and recovery support. We use DB\_ENV->lock\_vec to perform lock coupling, a technique that enhances the concurrency of Btree traversals [[Com79](http://www.aosabook.org/en/bib1.html#bib:comer:btree)]. In lock coupling, you hold one lock only long enough to acquire the next lock. That is, you lock an internal Btree page only long enough to read the information that allows you to select and lock a page at the next level.

虽然Berkeley DB内部没有采用层次封锁，它利用了这个能力来指定不同的冲突矩阵，以及一次性指定多个锁请求。在提供事务支持时，我们采用缺省的冲突矩阵；但采用另一个冲突矩阵以支持不带事务的简单的并发存取和恢复支持。我们采用DB\_ENV->lock\_vec来处理锁的耦合，这是一种增强B树遍历的并发性的技术[[Com79](http://www.aosabook.org/en/bib1.html#bib:comer:btree)]。在锁耦合中，你只用持有锁足够的时间以获取下一个锁。也就是说，你只需要锁住一个内部的B树页面足够长的时间以读到选择和锁定下一级页面的信息。

***Design Lesson 10 设计教训10***

Berkeley DB's general-purpose design was well rewarded when we added concurrent data store functionality. Initially Berkeley DB provided only two modes of operation: either you ran without any write concurrency or with full transaction support. Transaction support carries a certain degree of complexity for the developer and we found some applications wanted improved concurrency without the overhead of full transactional support. To provide this feature, we added support for API-level locking that allows concurrency, while guaranteeing no deadlocks. This required a new and different lock mode to work in the presence of cursors. Rather than adding special purpose code to the lock manager, we were able to create an alternate lock matrix that supported only the lock modes necessary for the API-level locking. Thus, simply by configuring the lock manager differently, we were able provide the locking support we needed. (Sadly, it was not as easy to change the access methods; there are still significant parts of the access method code to handle this special mode of concurrent access.)

Berkeley DB的通用设计在我们增加并发数据存储功能时获得了很好的回报。最初Berkeley DB只提供了两种操作模式：要么没有写并发性的运行，要么支持全部事务支持。事务支持给开发人员带来了一定程度的复杂性，我们发现有些应用程序想提高并发性又不想要全事务支持的额外代价。为了提供这个特性，我们增加了API级别的封锁以允许并发性，同时保证没有死锁。这需要一个新的和不同的封锁模式以支持游标。与其在锁管理器中增加特殊目的的代码，我们能够创建另外一种锁矩阵以支持API级锁定只需要的封锁模式。于是，仅仅通过将锁管理器配置得不同，我们就能提供我们需要的封锁支持。（不幸的是，修改存取方法就不那么容易了，存取方法中还有相当大的一部分代码要处理这种并发存取的特殊模式）

***4.8. The Log Manager: Log 日志管理器：Log***

The log manager provides the abstraction of a structured, append-only file. As with the other modules, we intended to design a general-purpose logging facility, however the logging subsystem is probably the module where we were least successful.

日志管理器提供了一个结构化的、仅限追加的文件的抽象。与其他模块一样，我们试图设计出一个通用的日志设施，然而日志子系统可能是我们做的最不成功的一个模块。

***Design Lesson 11 设计教训11***

When you find an architectural problem you don't want to fix "right now" and that you're inclined to just let go, remember that being nibbled to death by ducks will kill you just as surely as being trampled by elephants. Don't be too hesitant to change entire frameworks to improve software structure, and when you make the changes, don't make a partial change with the idea that you'll clean up later—do it all and then move forward. As has been often repeated, "If you don't have the time to do it right now, you won't find the time to do it later." And while you're changing the framework, write the test structure as well.

当你发现一个体系结构上的问题而又不想立即修复时，你其实倾向于放过它。请记住被蚕食而死和被大象踩住都一定会要你的命。别太犹豫而不去修改整个框架来改进软件结构，而且当你做出修改时，不要以为你以后会清理它而做出不完全的修改——一次做完并继续向前进。就像经常说的，“如果你现在没有时间去做，以后也不会有时间去做”。此外，在你修改框架时，同时也要写测试结构。

A log is conceptually quite simple: it takes opaque byte strings and writes them sequentially to a file, assigning each a unique identifier, called a log sequence number (LSN). Additionally, the log must provide efficient forward and backward traversal and retrieval by LSN. There are two tricky parts: first, the log must guarantee it is in a consistent state after any possible failure (where consistent means it contains a contiguous sequence of uncorrupted log records); second, because log records must be written to stable storage for transactions to commit, the performance of the log is usually what bounds the performance of any transactional application.

日志在概念上很简单：它拿到不透明的字节串并将它们顺序地写到文件中，给每笔日志一个称作日志顺序号（LSN）的唯一标识。此外，日志必须提供通过LSN的高效的正向和反向遍历和检索。这里有两个需要慎重处理的地方：第一，日志必须要保证在任何可能的故障后处于一个一致的状态（这里“一致”指的是未损坏的日志记录的连续序列）；其次，因为日志记录被写到稳定存储中以支持事务的提交，日志的性能通常会限定事务性应用的性能。

As the log is an append-only data structure, it can grow without bound. We implement the log as a collection of sequentially numbered files, so log space may be reclaimed by simply removing old log files. Given the multi-file architecture of the log, we form LSNs as pairs specifying a file number and offset within the file. Thus, given an LSN, it is trivial for the log manager to locate the record: it seeks to the given offset of the given log file and returns the record written at that location. But how does the log manager know how many bytes to return from that location?

因为日志是一个仅限追加的数据结构，它可能会无限制增长。我们把日志实现为一组顺序编号的文件，因此，日志空间可以通过简单的删除旧日志文件来回收。在这种多文件的日志结构下，我们把LSN定义为文件号和文件内偏移组成的对。于是，给定一个LSN，日志管理器定位日志记录就很简单了：它移动到给定日志文件的给定偏移，并返回该位置的日志记录。但是日志管理器怎么知道从该位置返回多少字节呢？

***4.8.1. Log Record Formatting 日志记录格式***

The log must persist per-record metadata so that, given an LSN, the log manager can determine the size of the record to return. At a minimum, it needs to know the length of the record. We prepend every log record with a log record header containing the record's length, the offset of the previous record (to facilitate backward traversal), and a checksum for the log record (to identify log corruption and the end of the log file). This metadata is sufficient for the log manager to maintain the sequence of log records, but it is not sufficient to actually implement recovery; that functionality is encoded in the contents of log records and in how Berkeley DB uses those log records.

日志必须保留每个日志记录的元数据以保证给定一个LSN，日志管理器可以判断待返回的记录的大小。至少，它需要知道日志记录的长度。我们假定每个日志记录都有一个包含记录长度的日志记录头、前一个日志记录的偏移位置（以支持反向遍历），以及一个日志记录的校验和（以标识日志的损坏和日志文件的结束）。这些元数据足够让日志管理器维护日志记录的顺序了，但是这还不足以支持恢复的实现；该功能要靠日志记录中的内容以及Berkeley DB怎么用这些日志记录来实现。

Berkeley DB uses the log manager to write before- and after-images of data before updating items in the database [[HR83](http://www.aosabook.org/en/bib1.html#bib:haerder:recovery)]. These log records contain enough information to either redo or undo operations on the database. Berkeley DB then uses the log both for transaction abort (that is, undoing any effects of a transaction when the transaction is discarded) and recovery after application or system failure.

Berkeley DB通过日志管理器在数据库中更新数据项前写下数据的前像和后像[[HR83](http://www.aosabook.org/en/bib1.html#bib:haerder:recovery)]。这些日志记录包含了重做或撤销数据库中操作的足够信息。Berkeley DB利用这些信息处理事务撤销（即，在事务撤销时撤销该事务的所有影响）和应用故障或系统故障后的恢复。

In addition to APIs to read and write log records, the log manager provides an API to force log records to disk (DB\_ENV->log\_flush). This allows Berkeley DB to implement write-ahead logging—before evicting a page from Mpool, Berkeley DB examines the LSN on the page and asks the log manager to guarantee that the specified LSN is on stable storage. Only then does Mpool write the page to disk.

除了读写日志记录的API之外，日志管理器还提供了一个强制将日志记录刷出到磁盘的API（DB\_ENV->log\_flush）。该API允许Berkeley DB实现WAL——在Mpool中回收页面前，Berkeley DB检查页面的LSN并且要求日志管理器保证该LSN已经在稳定存储上了。只有这样，Mpool才会将页面写到磁盘。

***Design Lesson 12 设计教训12***

Mpool and Log use internal handle methods to facilitate write-ahead logging, and in some cases, the method declaration is longer than the code it runs, since the code is often comparing two integral values and nothing more. Why bother with such insignificant methods, just to maintain consistent layering? Because if your code is not so object-oriented as to make your teeth hurt, it is not object-oriented enough. Every piece of code should do a small number of things and there should be a high-level design encouraging programmers to build functionality out of smaller chunks of functionality, and so on. If there's anything we have learned about software development in the past few decades, it is that our ability to build and maintain significant pieces of software is fragile. Building and maintaining significant pieces of software is difficult and error-prone, and as the software architect, you must do everything that you can, as early as you can, as often as you can, to maximize the information conveyed in the structure of your software.

Mpool和Log用内部的处理方法来处理WAL，在某些情况下，方法的声明比本身的代码还要长，因为代码除了比较两个整数之外什么也不做。为什么弄这些不太重要的方法仅仅去维护一致的层次呢？因为如果你的代码不是面向对象到了让你牙疼的话，它还不够面向对象。每段代码应该做少量的事情并且应该有个鼓励程序员在小的功能块之上构建新功能的上层设计。如果说我们在过去的几十年中学到了什么软件开发的东西的话，那就是我们构建和维护大量软件的能力是很弱的。构建和维护大量软件是困难和容易出错的，作为软件架构师，你必须尽你所能、尽早、尽量频繁的最大化软件结构表达的信息。

Berkeley DB imposes structure on the log records to facilitate recovery. Most Berkeley DB log records describe transactional updates. Thus, most log records correspond to page modifications to a database, performed on behalf of a transaction. This description provides the basis for identifying what metadata Berkeley DB must attach to each log record: a database, a transaction, and a record type. The transaction identifier and record type fields are present in every record at the same location. This allows the recovery system to extract a record type and dispatch the record to an appropriate handler that can interpret the record and perform appropriate actions. The transaction identifier lets the recovery process identify the transaction to which a log record belongs, so that during the various stages of recovery, it knows whether the record can be ignored or must be processed.

Berkeley DB在日志记录上施以结构以减少恢复的难度。大部分Berkeley DB的日志记录描述的是事务性更新。也就是说，大部分日志记录对应于以事务身份所做的数据库页面更新。这个描述有助于我们识别哪些是Berkeley DB必须附加到每条日志记录的元数据：数据库、事务和记录类型。事务标识和记录类型字段在每个记录的同一位置出现。这使得恢复系统可以抽取出日志类型并且将记录分发到可以解释和执行相关动作的合适的处理者。事务标识让恢复过程识别日志记录属于哪个事务，使得在恢复的不同阶段中，它知道该记录是否可以被忽略还是必须被处理。

***4.8.2. Breaking the Abstraction 打破抽象层***

There are also a few "special" log records. Checkpoint records are, perhaps, the most familiar of those special records. Checkpointing is the process of making the on-disk state of the database consistent as of some point in time. In other words, Berkeley DB aggressively caches database pages in Mpool for performance. However, those pages must eventually get written to disk and the sooner we do so, the more quickly we will be able to recover in the case of application or system failure. This implies a trade-off between the frequency of checkpointing and the length of recovery: the more frequently a system takes checkpoints, the more quickly it will be able to recover. Checkpointing is a transaction function, so we'll describe the details of checkpointing in the next section. For the purposes of this section, we'll talk about checkpoint records and how the log manager struggles between being a stand-alone module and a special-purpose Berkeley DB component.

还有一些“特殊的”日志记录。检查点记录可能是这些特殊记录中最熟悉的。做检查点是使数据库的磁盘状态在某个时间点一致的过程。换句话说，Berkeley DB为了性能尽量将数据库页缓存在Mpool中。然而，这些页面最终必须被写到磁盘，而我们越早做这个，在应用或系统故障时我们就能更快得恢复。这意味着需要在做检查点的频率和恢复时间长短之间权衡：系统做检查点越频繁，它就能更快得恢复。做检查点是一个事务性功能，因为我们将在下一节介绍它的细节。就本节而言，我们将谈谈检查点记录以及日志管理器如何在成为一个独立的模块和一个专用的Berkeley DB组件之间挣扎的。

In general, the log manager, itself, has no notion of record types, so in theory, it should not distinguish between checkpoint records and other records—they are simply opaque byte strings that the log manager writes to disk. In practice, the log maintains metadata revealing that it does understand the contents of some records. For example, during log startup, the log manager examines all the log files it can find to identify the most recently written log file. It assumes that all log files prior to that one are complete and intact, and then sets out to examine the most recent log file and determine how much of it contains valid log records. It reads from the beginning of a log file, stopping if/when it encounters a log record header that does not checksum properly, which indicates either the end of the log or the beginning of log file corruption. In either case, it determines the logical end of log.

总之，日志管理器本身没有记录类型的概念，因此在理论上，它不需要区分检查点记录和其他的记录——它们都仅仅是需要日志管理器写到磁盘的不透明字节串。实际上，日志管理器维护了元数据，说明了它确实理解一些记录的内容。例如，在日志启动过程中，日志管理器检查所有它能找到的日志文件并且识别出最近写过的日志文件。它假定所有该文件之前的所有日志文件都是完整无缺的，然后开始检查最近的日志文件并确定它含有哪些有效的记录。它从日志文件的开头开始读，直到遇到一个不能正确校验的日志记录头才停下来，这意味着到了日志尾或日志文件损坏了。这两种情况都确定了日志的逻辑结尾。

During this process of reading the log to find the current end, the log manager extracts the Berkeley DB record type, looking for checkpoint records. It retains the position of the last checkpoint record it finds in log manager metadata as a "favor" to the transaction system. That is, the transaction system needs to find the last checkpoint, but rather than having both the log manager and transaction manager read the entire log file to do so, the transaction manager delegates that task to the log manager. This is a classic example of violating abstraction boundaries in exchange for performance.

在读取日志以找到当前日志尾的过程中，日志管理器抽取Berkeley DB的记录类型以寻找检查点记录。作为对事务系统的“帮忙”，它把找到的最后一个检查地点记录的位置保留在日志管理器的元数据中。也就是说，事务系统需要找到最后的检查点，但是与其让日志管理器和事务管理器都去读取整个日志来干这件事，事务管理器把该任务代理给了日志管理器。这是一个违背抽象边界而换来性能的典型例子。

What are the implications of this tradeoff? Imagine that a system other than Berkeley DB is using the log manager. If it happens to write the value corresponding to the checkpoint record type in the same position that Berkeley DB places its record type, then the log manager will identify that record as a checkpoint record. However, unless the application asks the log manager for that information (by directly accessing cached\_ckp\_lsn field in the log metadata), this information never affects anything. In short, this is either a harmful layering violation or a savvy performance optimization.

这个权衡意味着什么呢？假设Berkeley DB之外有个系统在使用日志管理器。如果它碰巧写了一个检查点日志类型对应的值到了Berkeley DB放置自己的记录类型的同一个位置，那么日志管理器将把该记录识别为一个检查点记录。然而，除非应用程序找日志管理要这些信息（通过直接读取日志元数据中的cached\_ckp\_lsn字段），这些信息不会影响任何事情。简而言之，这既不是一个有害的对分层的违背，也不是一个精明的性能优化。

File management is another place where the separation between the log manager and Berkeley DB is fuzzy. As mentioned earlier, most Berkeley DB log records have to identify a database. Each log record could contain the full filename of the database, but that would be expensive in terms of log space, and clumsy, because recovery would have to map that name to some sort of handle it could use to access the database (either a file descriptor or a database handle). Instead, Berkeley DB identifies databases in the log by an integer identifier, called a log file id, and implements a set of functions, called dbreg (for "database registration"), to maintain mappings between filenames and log file ids. The persistent version of this mapping (with the record type DBREG\_REGISTER) is written to log records when the database is opened. However, we also need in-memory representations of this mapping to facilitate transaction abort and recovery. What subsystem should be responsible for maintaining this mapping?

文件管理部分是日志管理器与Berkeley DB其他模块间的分离比较模糊的另一个例子。就像前面提到的一样，大部分Berkeley DB的日志记录需要标识一个数据库。每条日志记录都可能包含数据库的全名，但这样在日志空间的角度看将是很昂贵的，也比较难看，因为恢复将需要把这个名字映射到某种形式的句柄以便能够访问数据库（要么是一个文件描述符要么是一个数据库句柄）。实际上，Berkeley DB在日志中用一个整数标识数据库，称为一个日志文件ID，并实现了一系列的函数，统称为dbreg（database registration的简称），来维护文件名和日志文件ID的映射。当数据库被打开时，这个映射的持久化版本（记录类型为DBREG\_REGISTER）被写到日志记录中。然而，我们也需要这个映射的内存表示以支持事务的撤销和恢复。哪个子系统应该负责维护这个映射呢？

In theory, the file to log-file-id mapping is a high-level Berkeley DB function; it does not belong to any of the subsystems, which were intended to be ignorant of the larger picture. In the original design, this information was left in the logging subsystems data structures because the logging system seemed like the best choice. However, after repeatedly finding and fixing bugs in the implementation, the mapping support was pulled out of the logging subsystem code and into its own small subsystem with its own object-oriented interfaces and private data structures. (In retrospect, this information should logically have been placed with the Berkeley DB environment information itself, outside of any subsystem.)

理论上，文件到日志文件ID的映射是一个高层的Berkeley DB函数；它不属于任何一个子系统，子系统不应有全局概念。在最初的设计中，这些信息被留在日志子系统的数据结构中，因为日志系统看起来是最好的选择。然而，在不断地发现和修复实现中的缺陷时，这个映射支持被从日志子系统代码中抽取出来形成了它自己小子系统，有了自己的面向对象的接口和私有的数据结构。（回过来看，这些信息逻辑上本应该被放在Berkeley DB环境信息本身中，在所有子系统之外。）

***Design Lesson 13 设计教训13***

There is rarely such thing as an unimportant bug. Sure, there's a typo now and then, but usually a bug implies somebody didn't fully understand what they were doing and implemented the wrong thing. When you fix a bug, don't look for the symptom: look for the underlying cause, the misunderstanding, if you will, because that leads to a better understanding of the program's architecture as well as revealing fundamental underlying flaws in the design itself.

极少存在“不重要的Bug”这样的事情。确实，不时会有一些笔误，但通常一个Bug意味着有人没有完全理解他们在做的事情并实现错了。当你修复Bug时，不要仅看现象，要看底层的原因。如果你愿意的话，还应该看看产生误解的原因，因为这样可以更好的理解程序的体系结构并发现设计本身更本质的缺陷。

***4.9. The Transaction Manager: Txn 事务管理器：Txn***

Our last module is the transaction manager, which ties together the individual components to provide the transactional ACID properties of atomicity, consistency, isolation, and durability. The transaction manager is responsible for beginning and completing (either committing or aborting) transactions, coordinating the log and buffer managers to take transaction checkpoints, and orchestrating recovery. We'll visit each of these areas in order.

我们的最后一个模块是事务管理器，它把各个独立的模块联系在一起以提供事务的ACID属性（原子性、一致性、隔离性和持久性）。事务管理器负责事务的开始和结束（要么提交，要么撤销），协调日志管理器和缓冲区管理器做事务检查点并组织恢复。我们将按顺序逐一讨论这些领域。

Jim Gray invented the ACID acronym to describe the key properties that transactions provide [[Gra81](http://www.aosabook.org/en/bib1.html#bib:gray:trans)]. Atomicity means that all the operations performed within a transaction appear in the database in a single unit—they either are all present in the database or all absent. Consistency means that a transaction moves the database from one logically consistent state to another. For example, if the application specifies that all employees must be assigned to a department that is described in the database, then the consistency property enforces that (with properly written transactions). Isolation means that from the perspective of a transaction, it appears that the transaction is running sequentially without any concurrent transactions running. Finally, durability means that once a transaction is committed, it stays committed—no failure can cause a committed transaction to disappear.

Jim Gray发明了ACID这个缩写词来描述事务提供的关键属性 [[Gra81](http://www.aosabook.org/en/bib1.html#bib:gray:trans)] 。原子性的意思是一个事务中执行的所有操作在数据库中表现为一个单一的单元——它们要么都在数据库中，要么都不在。一致性的意思是事务把数据库从一个逻辑一致的状态转移到另一个。例如，如果应用程序要求每个员工都必须被安排到一个已在数据库中定义了的部门，那么一致性属性将会确保它（在事务正确书写时）。隔离性的意思是从每个事务的角度看，它就像是在没有任何其他并发事务在运行时顺序执行的。最后，持久性的意思是一旦事务被提交，它就保持提交状态——没有故障可以使得已经提交的事务消失掉。

The transaction subsystem enforces the ACID properties, with the assistance of the other subsystems. It uses traditional transaction begin, commit, and abort operations to delimit the beginning and ending points of a transaction. It also provides a prepare call, which facilitates two phase commit, a technique for providing transactional properties across distributed transactions, which are not discussed in this chapter. Transaction begin allocates a new transaction identifier and returns a transaction handle, DB\_TXN, to the application. Transaction commit writes a commit log record and then forces the log to disk (unless the application indicates that it is willing to forego durability in exchange for faster commit processing), ensuring that even in the presence of failure, the transaction will be committed. Transaction abort reads backwards through the log records belonging to the designated transaction, undoing each operation that the transaction had done, returning the database to its pre-transaction state.

事务子系统在其他子系统的协助下确保ACID属性。它采用传统的事务开始、提交和撤销操作来分隔事务的开始点和结束点。它也提供了一个prepare调用以实现两阶段提交，两阶段提交是在分布事务之间提供事务属性的技术，本章对此没有描述。事务开始要分配一个新的事务标识符并返回一个事务句柄DB\_TXN给应用程序。事务提交要写一个提交日志记录然后强制刷出日志到磁盘（除非应用程序表明它愿意放弃持久性以换取更快的提交处理），保证即使在出现故障时，事务也会被提交。事务撤销会反向读取属于对应事务的日志记录，撤销该事务已经做的每个操作，将数据库退回到该事务开始前的状态。

***4.9.1. Checkpoint Processing 检查点处理***

The transaction manager is also responsible for taking checkpoints. There are a number of different techniques in the literature for taking checkpoints [[HR83](http://www.aosabook.org/en/bib1.html#bib:haerder:recovery)]. Berkeley DB uses a variant of fuzzy checkpointing. Fundamentally, checkpointing involves writing buffers from Mpool to disk. This is a potentially expensive operation, and it's important that the system continues to process new transactions while doing so, to avoid long service disruptions. At the beginning of a checkpoint, Berkeley DB examines the set of currently active transactions to find the lowest LSN written by any of them. This LSN becomes the checkpoint LSN. The transaction manager then asks Mpool to flush its dirty buffers to disk; writing those buffers might trigger log flush operations. After all the buffers are safely on disk, the transaction manager then writes a checkpoint record containing the checkpoint LSN. This record states that all the operations described by log records before the checkpoint LSN are now safely on disk. Therefore, log records prior to the checkpoint LSN are no longer necessary for recovery. This has two implications: First, the system can reclaim any log files prior to the checkpoint LSN. Second, recovery need only process records after the checkpoint LSN, because the updates described by records prior to the checkpoint LSN are reflected in the on-disk state.

事务管理器也负责做检查点。在学术界有很多不同的技术来做检查点 [[HR83](http://www.aosabook.org/en/bib1.html#bib:haerder:recovery)] 。Berkeley DB采用了模糊检查点的一个变种。从根本上看，做检查点需要需要将缓冲区从Mpool中写到磁盘。这是一个很可能代价昂贵的操作，重要的是系统同时能继续处理新的事务以避免长时间的服务中断。在检查点开始时，Berkeley DB检查当前活动的事务集合以找到它们当中任何一个所写的最小的LSN。该LSN就是检查点LSN。事务管理器然后要求Mpool去刷新缓存中的脏页到磁盘，写这些缓冲可能会触发日志的刷出操作。在所有这些缓冲都被安全写到磁盘后，事务管理器会写一个包含检查点LSN的检查点记录。该记录表明在检查点LSN之前的日志记录描述的所有操作现在都安全的存在磁盘上了。因此，在检查点LSN之前的日志记录就不再需要用来恢复了。这有两重意思：第一，系统可以回收检查点LSN之前的任意日志文件。第二，恢复只需要处理检查点LSN之后的日志记录。因为检查点LSN之前的日志记录所描述的更新已经被反映在磁盘状态中了。

Note that there may be many log records between the checkpoint LSN and the actual checkpoint record. That's fine, since those records describe operations that logically happened after the checkpoint and that may need to be recovered if the system fails.

注意在检查点LSN和实际的检查点记录之间可能存在很多的日志记录。这没什么问题，因为那些记录描述的操作逻辑上发生在检查点之后，因此如果系统故障了还是需要做恢复的。

***4.9.2. Recovery 恢复***

The last piece of the transactional puzzle is recovery. The goal of recovery is to move the on-disk database from a potentially inconsistent state to a consistent state. Berkeley DB uses a fairly conventional two-pass scheme that corresponds loosely to "relative to the last checkpoint LSN, undo any transactions that never committed and redo any transactions that did commit." The details are a bit more involved.

事务性难题的最后一个部分是恢复。恢复的目标是将磁盘数据库从一个可能不一致的状态转到一个一致的状态。Berkeley DB采用一个相当传统的两遍模式，大致对应于“相对于最后的检查点LSN，撤销所有没有提交的事务并重做所有已经提交的事务”。后面将介绍更多的细节。

Berkeley DB needs to reconstruct its mapping between log file ids and actual databases so that it can redo and undo operations on the databases. The log contains a full history of DBREG\_REGISTER log records, but since databases stay open for a long time and we do not want to require that log files persist for the entire duration a database is open, we'd like a more efficient way to access this mapping. Prior to writing a checkpoint record, the transaction manager writes a collection of DBREG\_REGISTER records describing the current mapping from log file ids to databases. During recovery, Berkeley DB uses these log records to reconstruct the file mapping.

Berkeley DB需要重新构造从日志文件ID到实际的数据库之间的映射以便它可以重做和撤销数据库中的操作。日志中包含了DBREG\_REGISTER日志记录的完整历史，但是数据库会长时间处于打开状态，我们不想保留整个数据库打开期间的日志文件，而需要一个更有效的方法来访问这个映射。在写检查点记录前，事务管理器写下一组DBREG\_REGISTER记录来描述当前的从日志文件ID到数据库的映射。在恢复期间，Berkeley DB使用这些日志记录去重新构造文件映射。

When recovery begins, the transaction manager probes the log manager's cached\_ckp\_lsn value to determine the location of the last checkpoint record in the log. This record contains the checkpoint LSN. Berkeley DB needs to recover from that checkpoint LSN, but in order to do so, it needs to reconstruct the log file id mapping that existed at the checkpoint LSN; this information appears in the checkpoint *prior* to the checkpoint LSN. Therefore, Berkeley DB must look for the last checkpoint record that occurs before the checkpoint LSN. Checkpoint records contain, not only the checkpoint LSN, but the LSN of the previous checkpoint to facilitate this process. Recovery begins at the most recent checkpoint and using the prev\_lsn field in each checkpoint record, traverses checkpoint records backwards through the log until it finds a checkpoint record appearing before the checkpoint LSN. Algorithmically:

当恢复开始时，事务管理器检查日志管理器的cached\_ckp\_lsn值来判断最后一个检查点记录的位置。该记录包含检查点的LSN。Berkeley DB需要从该检查点LSN开始恢复，但是为了做这件事，它需要重新构造在该检查点LSN处的日志文件ID映射；这些信息在该检查点LSN之前的检查点记录中。因此，Berkeley DB必须查找在该检查点LSN之前的最近一个检查点记录。检查点记录不仅包含了检查点LSN，还有前一个检查点的LSN（以支持这个查找过程）。恢复从最近的检查点开始，采用每个检查点记录中的prev\_lsn字段去反向遍历日志直到它找到了一个出现在检查点LSN之前的检查点记录。算法如下：

ckp\_record = read (cached\_ckp\_lsn)

ckp\_lsn = ckp\_record.checkpoint\_lsn

cur\_lsn = ckp\_record.my\_lsn

while (cur\_lsn > ckp\_lsn) {

ckp\_record = read (ckp\_record.prev\_ckp)

cur\_lsn = ckp\_record.my\_lsn

}

Starting with the checkpoint selected by the previous algorithm, recovery reads sequentially until the end of the log to reconstruct the log file id mappings. When it reaches the end of the log, its mappings should correspond exactly to the mappings that existed when the system stopped. Also during this pass, recovery keeps track of any transaction commit records encountered, recording their transaction identifiers. Any transaction for which log records appear, but whose transaction identifier does not appear in a transaction commit record, was either aborted or never completed and should be treated as aborted. When recovery reaches the end of the log, it reverses direction and begins reading backwards through the log. For each transactional log record encountered, it extracts the transaction identifier and consults the list of transactions that have committed, to determine if this record should be undone. If it finds that the transaction identifier does not belong to a committed transaction, it extracts the record type and calls a recovery routine for that log record, directing it to undo the operation described. If the record belongs to a committed transaction, recovery ignores it on the backwards pass. This backward pass continues all the way back to the checkpoint LSN[1](http://www.aosabook.org/en/bdb.html#footnote-1). Finally, recovery reads the log one last time in the forward direction, this time redoing any log records belonging to committed transactions. When this final pass completes, recovery takes a checkpoint. At this point, the database is fully consistent and ready to begin running the application.

从前面的算法找到的检查点开始，恢复算法顺序读取到日志尾以重新构造日志文件ID映射。当它到达日志尾时，映射应该准确的对应系统停止时存在的映射。也是在这个阶段中，恢复算法跟踪遇到的每个事务提交记录，记录它们的事务标识。所有有日志记录但是其日志标识未在事务提交记录中出现的事务要么是被回滚了，要么是从未完成从而也应被视为回滚了。当恢复到日志尾时，它调转方向并开始反向读取日志。对于遇到的每个事务日志记录，它抽取出事务标识并构造出已经提交事务的列表，以决定该记录是否该被回滚。如果它找到事务标识不属于提交的事务，就抽取出记录类型并且调用一个该日志记录的恢复例程，指导它去撤销对应的操作。如果该记录属于一个已提交的事务，恢复在反向扫描时忽略它。反向扫描一直进行到检查点LSN[1](http://www.aosabook.org/en/bdb.html#footnote-1)（Notes：这里有个脚注）。最终，恢复再以正向方式最后读一遍日志，这次重做所有属于已提交事务的日志记录。当这最后一个阶段完成时，恢复做一个检查点。此时，数据库完全一致了，可以开始运行应用程序了。

Thus, recovery can be summarized as:

总之，恢复可以被总结为：

1. Find the checkpoint prior to the checkpoint LSN in the most recent checkpoint
2. Read forward to restore log file id mappings and construct a list of committed transactions
3. Read backward to the checkpoint LSN, undoing all operations for uncommitted transactions
4. Read forward, redoing all operations for committed transactions
5. Checkpoint
6. 找到最近检查点记录中检查点LSN之前的那个检查点
7. 正向读取日志以恢复日志文件ID映射并构造出已提交事务的列表
8. 反向读取日志到检查点LSN，撤销未提交事务的所有操作
9. 正向读取日志，重做已提交事务的所有操作
10. 做检查点

In theory, the final checkpoint is unnecessary. In practice, it bounds the time for future recoveries and leaves the database in a consistent state.

理论上，最后一个检查点是不必要的。实际上，它减少了未来的恢复的时间并使得数据库处于一个一致的状态。

***Design Lesson 14 设计教训14***

Database recovery is a complex topic, difficult to write and harder to debug because recovery simply shouldn't happen all that often. In his Turing Award Lecture, Edsger Dijkstra argued that programming was inherently difficult and the beginning of wisdom is to admit we are unequal to the task. Our goal as architects and programmers is to use the tools at our disposal: design, problem decomposition, review, testing, naming and style conventions, and other good habits, to constrain programming problems to problems we *can* solve.

数据库恢复是一个复杂的主题，很难写，更难调试，因为恢复根本不会频繁的发生。在他的图灵奖演讲中，Edsger Dijkstra认为编程天生很难，承认我们不擅此道是智慧的开端。我们作为架构师和程序员的目的是使用我们所掌握的工具：设计、问题分解、评审、测试、命名和风格规范以及其它好的习惯来限制编程问题为我们能解决的问题。

***4.10. Wrapping Up 结束语***

Berkeley DB is now over twenty years old. It was arguably the first general-purpose transactional key/value store and is the grandfather of the NoSQL movement. Berkeley DB continues as the underlying storage system for hundreds of commercial products and thousands of Open Source applications (including SQL, XML and NoSQL engines) and has millions of deployments across the globe. The lessons we've learned over the course of its development and maintenance are encapsulated in the code and summarized in the design tips outlined above. We offer them in the hope that other software designers and architects will find them useful.

Berkeley DB现在已经年满20岁了。它可以说是第一个通用的事务性键/值存储，也是NoSQL运动的鼻祖。Berkeley DB继续作为几百个商业产品和几千个开源应用软件（包括SQL、XML和NoSQL引擎）的底层存储系统，并在全球有几百万个部署。我们在它的开发和维护过程中所学到的经验教训都体现在代码和上面总结的设计提示中了。我们分享并希望其他的软件设计者和架构师发现它们有用。

***Footnotes 脚注***

1. Note that we only need to go backwards to the checkpoint LSN, not the checkpoint record preceding it.  
   请注意我们只需要返回到检查点LSN，而不是它前面的检查点记录。